LIS Developer's Guide

Submitted under Task Agreement GSFC-CT-2

Cooperative Agreement Notice (CAN) CAN-00OES-01

Increasing Interoperability and Performance of
Grand Challenge Applications in the Earth, Space, Life, and Microgravity Sciences

May 7, 2004

Version 3.0

History:

Revision	Summary of Changes	Date
3.0	Milestone "G" submission	May 7, 2004
2.3	LIS 2.3 code release	December 19, 2003
	Initial revison	

National Aeronautics and Space Administration
Goddard Space Flight Center
Greenbelt, Maryland 20771

Contents

1 Introduction

This document describes some of the interoperable features in LIS and how to use/extend them. The following sections describe the general development and documentation practices recommended for using and extending LIS software, followed by the guidelines for using the extensible features in LIS for customization and improved functionality.

2 Background

This section provides some general information about the LIS project and land surface modeling.

2.1 LIS

The primary goal of the LIS project is to build a system that is capable of performing high resolution land surface modeling at high performance using scalable computing technologies. The LIS software system consists of a number of components: (1) LIS driver: the core software that integrates the use of land surface models, data management techniques, and high performance computing. (2) community land surface models such as CLM [2], Noah [4], and VIC [7], and (3) Visualization and data management tools such as GrADS [3] -DODS [6] server. One of the important design goals of LIS is to develop an interoperable system to interface and interoperate with land surface modeling community and other earth system models. LIS is designed using an object oriented, component-based style. The adaptable interfaces in LIS can be used by the developers to ease the cost of development and foster rapid prototyping and development of applications. The following sections describe the main components of LIS.

2.2 LIS driver

The core of LIS software system is the LIS driver that controls program execution. The LIS driver is a model control and input/output system (consisting of a number of subroutines, modules written in Fortran 90 source code) that drives multiple offline one-dimensional LSMs. The one-dimensional LSMs such as CLM and Noah, apply the governing equations of the physical processes of the soil-vegetation-snowpack medium. These land surface models aim to characterize the transfer of mass, energy, and momentum between a vegetated surface and the atmosphere. When there are multiple vegetation types inside a grid box, the grid box is further divided into ”tiles”, with each tile representing a specific vegetation type within the grid box, in order to simulate sub-grid scale variability.

The execution of the LIS driver starts with reading in the user specifications, including the modeling domain, spatial resolution, duration of the run, etc. The LIS User’s Guide describes the exhaustive list of parameters specified by the user. This is followed by the reading and computing of model parameters. The time loop begins and forcing data is read, time/space interpolation is computed and modified as necessary. Forcing data is used to specify the boundary conditions to the land surface model. The LIS driver applies time/space interpolation to convert the forcing data to the appropriate resolution required by the model. The selected model is run for a vector of “tiles” and output and restart files are written at the specified output interval.

Some of the salient features provided by the LIS driver include:

- Vegetation type-based “tile” or “patch” approach to simulate sub-grid scale variability.
- Makes use of various satellite and ground-based observational systems.
- Derives model parameters from existing topography, vegetation, and soil coverages.
- Extensible interfaces to facilitate incorporation of new land surface models, forcing schemes.
- Uses a modular, object oriented style design that allows “plug and play” of different features by allowing user to select only the components of interest while building the executable.
- Ability to perform regional modeling (only on the domain of interest).
- Provides a number of scalable parallel processing modes of operation.

Please refer to the software design document for a detailed description of the design of LIS driver. The LIS developer’s guide describes how to use the extensible interfaces in LIS. The “plug and play” feature of different components is described in this document.

2.3 Community Land Model (CLM)

CLM (Community Land Model) is a 1-D land surface model, written in Fortran 90, developed by a grass-roots collaboration of scientists who have an interest in making a general land model available for public use. LIS currently uses CLM version 2.0. CLM version 2.0 was released in May 2002. The source code for CLM 2.0 is freely available from the National Center for Atmospheric Research (NCAR) [2]. The CLM is used as the land model for the Community Climate System Model (CCSM) (http://www.ccsm.ucar.edu/), which includes the Community Atmosphere Model (CAM) (http://www.cgd.ucar.edu/cms/). CLM is executed with all forcing, parameters, dimensioning, output routines, and coupling performed by an external driver of the user’s design (in this case done by LIS). CLM requires pre-processed data such as the land surface type, soil and vegetation parameters, model initialization, and atmospheric boundary conditions as input. The model applies finite-difference spatial discretization methods and a fully implicit time-integration scheme to numerically integrate the governing equations. The model subroutines apply the governing equations of the physical processes of the soil-vegetation-snowpack medium, including the surface energy balance equation, Richards’ [12] equation for soil hydraulics, the diffusion equation for soil heat transfer, the energy-mass balance equation for the snowpack, and the Collatz et al. [9] formulation for the conductance of canopy transpiration.

2.4 The Community Noah Land Surface Model

The community Noah Land Surface Model is a stand-alone, uncoupled, 1-D column model freely available at the National Centers for Environmental Prediction (NCEP; [4]). The name is an acronym representing the various developers of the model (N: NCEP; O: Oregon State University, Dept. of Atmospheric Sciences; A: Air Force (both AFWA and AFRL - formerly AFGL, PL); and H: Hydrologic Research Lab - NWS (now Office of Hydrologic Development – OHD)). Noah can be executed in either coupled or uncoupled mode. It has been coupled with the operational NCEP mesoscale Eta model [10] and its companion Eta Data Assimilation System (EDAS) [13], and the NCEP Global Forecast System (GFS) and its companion Global Data Assimilation System (GDAS). When Noah is executed in uncoupled mode, near-surface atmospheric forcing data (e.g., precipitation, radiation, wind speed, temperature, humidity) is required as input. Noah simulates soil moisture (both liquid and frozen), soil temperature, skin temperature, snowpack depth, snowpack water equivalent, canopy water content, and the energy flux and water flux terms of the surface energy balance and surface water balance. The model applies finite-difference spatial discretization methods and a Crank-Nicholson time-integration scheme to numerically integrate the governing equations of the physical processes of the soil vegetation-snowpack medium, including the surface energy balance equation, Richards' [12] equation for soil hydraulics, the diffusion equation for soil heat transfer, the energy-mass balance equation for the snowpack, and the Jarvis [11] equation for the conductance of canopy transpiration.

2.5 Variable Infiltration Capacity (VIC) Model

Variable Infiltration Capacity (VIC) model is a macroscale hydrologic model, written in C, being developed at the University of Washington and Princeton University. The VIC code repository along with the model description and source code documentation is publicly available at the Princeton website [7]. VIC is used in macroscopic land use models such as SEA - BASINS (http://boto.ocean.washington.edu/seasia/intro.htm). VIC is a semi-distributed, grid-based hydrological model, which parameterizes the dominant hydrometeorological processes taking place at the land surface - atmospheric interface. The execution of VIC model requires preprocessed data such as precipitation, temperature, meteorological forcing, soil and vegetation parameters, etc. as input. The model uses three soil layers and one vegetation layer with energy and moisture fluxes exchanged between the layers. The VIC model represents surface and subsurface hydrologic processes on a spatially distributed (grid cell) basis. Partitioning grid cell areas to different vegetation classes can approximate sub-grid scale variation in vegetation characteristics. VIC models the processes governing the flux and storage of water and heat in each cell-sized system of vegetation and soil structure. The water balance portion of VIC is based on three concepts: 1) Division of grid-cell into fraction sub-grid vegetation coverages.

2) The variable infiltration curve for rainfall/runoff partitioning at the land surface.
3) A baseflow/deep soil moisture curve for lateral baseflow.

Water balance calculations are preformed at three soil layers and within a vegetation canopy. An energy balance is calculated at the land surface. A full description of algorithms in VIC can be found in the references listed at the VIC website.

2.6 GrADS-DODS Server

A GrADS-DODS Server (GDS) is a data server built upon the Grid Analysis and Display System (GrADS) and the Distributed Oceanographic Data System (DODS).

GrADS is an earth science data manipulation and visualization tool under development at the Center for Ocean-Land-Atmosphere Studies (COLA) (`http://http://grads.iges.org/cola.html`). See `http://grads.iges.org/grads/grads.html` for more detailed information about GrADS.

DODS, also called the Open source Project for a Network Data Access Protocol (OPeNDAP), is a protocol for serving data-sets stored in various formats over a network. See `http://www.unidata.ucar.edu/packages/dods/` for more detailed information about DODS.

A GDS may be used to provide the LIS driver with the forcing and input parameter data needed to run an LSM.

A GDS is an optional component of the LIS system. LIS may be run without using a GDS to access the forcing and input parameter data-sets. All necessary forcing and input parameter data-sets may be stored on locally-accessable hard-disks and read in directly by the LIS driver, provided the computer system has sufficient memory.

The intent of a GDS for the LIS project is to provide the LIS driver with subsets of the forcing and input parameter data-sets, so that large-scale, high-resolution domains may be broken-up/parallelized and processed across many compute-nodes of a Beowulf cluster.

3 Coding and Documentation Conventions

This section describes some of the coding and documentation conventions [1] that are helpful for developers of LIS.

3.1 Coding conventions

LIS is implemented using the Fortran 90 and C programming languages. Since different Fortran compilers parse source files differently depending on file extensions such as f, f77, F, f90, and F90, the task of porting code to different platforms is a hard process. Therefore, Fortran additions and contributions to LIS code are expected to be written using the Fortran 90 dialect, and the sources files must have an F90 extension. Some of the style guidelines followed in LIS are as follows:

- Preprocessor: C preprocessor (cpp) is used wherever the use of a language preprocessor is required. The Fortran compiler is assumed to have the ability to run the preprocessor as part of the compilation process. The preprocessing tokens are written in uppercase to distinguish them from the Fortran code.
- Loops: All loops in Fortran are structured using do-enddo constructs as opposed to numbered loops.
- Indentation: Code with nested if blocks and do loops are expected to be indented for readability.
- Modules: Modules must be named the same as the file in which they reside. This is enforced due to the fact that make programs build dependencies based on file names.
- Implicit none: All variables in different modules should be explicitly typed, and this should be enforced by the use of the "implicit none" statement.

3.2 Documentation conventions

LIS uses an in-line documentation system that allows users to create both web-browsable (html) and print-friendly(ps/pdf) documentation. Each function, subroutine, or module includes a prologue instrumented for use with the ProTex auto-documentation script [5]. The following examples describe the documentation templates used in LIS.

Templates for routines that are not internal to modules.

```
!-------------------------------------------------------------------------
!         NASA/GSFC Land Information Systems LIS 2.3
!-------------------------------------------------------------------------
!BOP
!
! !ROUTINE:
!
! !INTERFACE:
!
! !USES:
!
! !INPUT PARAMETERS:
!
! !OUTPUT PARAMETERS:
!
! !DESCRIPTION:
!
! !BUGS:
!
! !SEE ALSO:
!
! !SYSTEM ROUTINES:
!
! !FILES USED:
!
! !REVISION HISTORY:
!
!  27Jun02  Username Initial specification
!
!EOP
!-------------------------------------------------------------------------
!BOC
!EOC
```

Template for a module :

```
!-------------------------------------------------------------------------
!          NASA/GSFC Land Information Systems LIS 2.3
!-------------------------------------------------------------------------
!BOP
!
! !MODULE:
!
! !PUBLIC TYPES:
!
! !PUBLIC MEMBER FUNCTIONS:
!
! !PUBLIC DATA MEMBERS:
!
! !DESCRIPTION:
!
! !REVISION HISTORY:
!
!  27Jun02  Username Initial specification
!
!EOP
```

Template for a C file:

```
//-----------------------------------------------------------------------
//          NASA/GSFC Land Information Systems LIS 2.3
//-----------------------------------------------------------------------
//BOP
//
// !ROUTINE:
//
// !INTERFACE:
//
// !USES:
//
// !INPUT PARAMETERS:
//
// !OUTPUT PARAMETERS:
//
// !DESCRIPTION:
//
// !BUGS:
//
// !SEE ALSO:
//
// !SYSTEM ROUTINES:
//
// !FILES USED:
//
// !REVISION HISTORY:
//
//  27Jun02  Username Initial specification
//
//EOP
//-----------------------------------------------------------------------
//BOC
//EOC
```

4 Customizable Features in LIS

The LIS driver is designed with extensible interfaces for facilitating easy incorporation of new features into LIS. The LIS driver uses advanced features of the Fortran 90 programming language, which are especially suitable for object oriented programming. The object oriented style of design adopted in LIS enables the driver to provide well defined interfaces or "plug points" for enabling rapid prototyping and development of new features and applications into LIS.

The LIS driver includes a number of functional extensions including:

- land surface model: Interfaces for adding new land surface models.
- base forcing: Interfaces for adding new model forcing schemes.
- observed forcing: Interfaces for adding observational forcing products.
- domain: Using a user specified domain or a subdomain of interest.
- parameters: Specifying custom defined data-sets.

4.1 Function Tables

The modules in LIS are constructed using a component-based design, with each module/component representing a program segment that is functionally related. The customizable interfaces in LIS are designed using a number of virtual function tables and the actual delegation of the calls are done at run-time by resolving the function names from the table. The C language allows the capability to store functions, table them, and pass them as arguments. The Fortran 90 programming language allows passing of functions as arguments. By combining these features of both languages, LIS uses a complete set of operations with function pointers.

Figure 1 illustrates how the function tables work in LIS. A function is stored in the table typically by a `register` function, that simply stores the pointer to the function at the specified index. When the function needs to be accessed, a generic call is made which resolves into a specific call depending on the index specified. This type of implementation helps in defining generic calls in the driver and to include only the components of interest while compiling and building the executable. For simplicity, throughout this document the word "registry" is used to refer to a function table.

The LIS 3.0 source code available from the LIS website contains a number of subdirectories, which are organized as components. The top level organization of the source (*src*) is as follows:

4.2 Defining source directories for compilation

A file called *Filepath* in the *$WORKING/LIS/src/make* directory specifies all the source files that will be included during compilation. A sample *Filepath* is shown below.

Directory Name	Synopsis
driver	LIS driver routines
baseforcing	Routines to call model forcing methods
obsprecips	Routines to call observed precipitation products
obsrads	Routines to call observed radiation products
forcing-plugin	Routines that define registries for forcing schemes
lsm-plugin	Routines that defines registries for land surface models
lsms	Contains land surface model codes

```
../driver
../lsm-plugin
../forcing-plugin
../iplib
../baseforcing/geos
../baseforcing/gdas
../obsprecips/cmap
../obsrads/agrmet
../lsms/noah.2.6
../lsms/mosaic
../lsms/vic
../lsms/clm2
../lsms/clm2/main
../lsms/clm2/biogeophys
../lsms/clm2/biogeochem
../lsms/clm2/camclm_share
../lsms/clm2/csm_share
../lsms/clm2/riverroute
../lsms/clm2/ecosysdyn
```

4.3 Defining components while building the executable

As described earlier, the design of LIS allows users to define and include only the components of interest while building the executable. Since Fortran is not a truly object oriented language, this type of runtime polymorphism can only be simulated in software.

The LIS developers guide describes how new land surface models, forcing schemes, etc. can be included in LIS. This is achieved by allowing the user to specify the extensible interfaces such as the ones provided in *$WORKING/LIS/src/lsm-plugin* and *$WORKING/LIS/src/forcing-plugin* directories. Once the user specifies the components to be used in these interfaces, the *Filepath* directory can be modified to include only these components. For example, if a user is interested in running only one land surface model (say Noah), using only the GEOS forcing scheme, and no observational forcing products, the *Filepath* directory reduces to:

```
../driver
../lsm-plugin
../forcing-plugin
../iplib
../baseforcing/geos
../lsms/noah.2.6
```

The extensible interfaces need to be defined as follows:
The *lsm_plugin* method in *$WORKING/LIS/src/lsm-plugin/lsm_pluginMod.F90* needs to be defined as:

```
       subroutine lsm_plugin
         use noah_varder, only : noah_varder_ini
         external noah_main
         external noah_setup
         external noahrst
         external noah_output
         external noah_f2t
         external noah_writerst
         external noah_dynsetup

         call registerlsmini(1,noah_varder_ini)
         call registerlsmsetup(1,noah_setup)
         call registerlsmdynsetup(1,noah_dynsetup)
         call registerlsmrun(1,noah_main)
         call registerlsmrestart(1,noahrst)
         call registerlsmoutput(1,noah_output)
         call registerlsmf2t(1,noah_f2t)
         call registerlsmwrst(1,noah_writerst)
       end subroutine lsm_plugin
```

The *baseforcing_plugin* method in *$WORKING/LIS/src/forcing-plugin/baseforcing_pluginMod.F90* needs to be defined as:

```
  subroutine baseforcing_plugin
    use geosdomain_module
    external getgeos
    external time_interp_geos
    call registerget(2,getgeos)
    call registerdefnat(2,defnatgeos)
    call registertimeinterp(2,time_interp_geos)
  end subroutine baseforcing_plugin
```

The *precipforcing_plugin* method in *$WORKING/LIS/src/forcing-plugin/precipforcing_pluginMod.F90* and the *radforcing_plugin* method in *$WORKING/LIS/src/forcing-plugin/radforcing_pluginMod.F90* can be left empty as follows:

```
  subroutine precipforcing_plugin
  end subroutine precipforcing_plugin

  subroutine radforcing_plugin

  end subroutine radforcing_plugin
```

Similarly, different combinations of using the components can be implemented defining the interfaces appropriately and chosing the corresponding source files through the *Filepath* file.

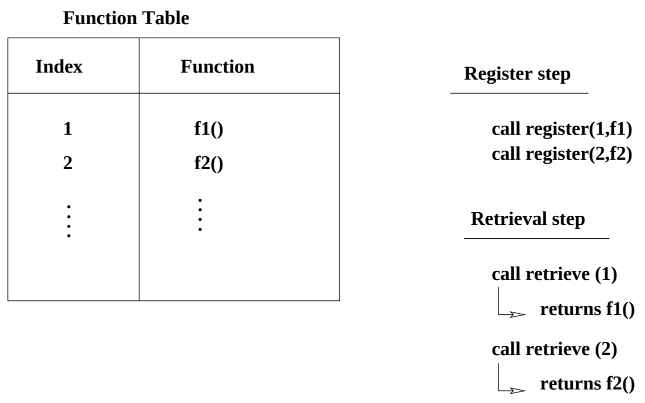

Figure 1: Example of a function table implementation

5 Customizing LIS to use new land surface models

The *lsm-plugin* directory contains the *lsm_pluginMod* module that can be used to customize and define land surface models in LIS. The *lsm_pluginMod* contains a *lsm_plugin* method that defines a number or registries to capture the basic offline operations of a land surface model. The registries can be used to define functions to perform the following tasks:

- initialization:
  Definition of land surface model variables, allocation of memory, reading run-time parameters, etc.
- setup:
  Initialization of land surface model parameters.
- dynamic setup:
  Routines to initialize or update time dependent parameters.
- run:
  Routines to execute land surface model on a single gridcell for a single timestep.
- write restart:
  Routines to write restart files
- read restart:
  Routines to read restart files
- output:
  Routines to write output
- transfer of forcing data to model tiles:
  Routines that provides an array of forcing variables for each gridcell.

The following example shows how the registry functions are defined for Noah land surface model.

```
call registerlsmini(1,noah_varder_ini)
call registerlsmsetup(1,noah_setup)
call registerlsmdynsetup(1,noah_dynsetup)
call registerlsmrun(1,noah_main)
call registerlsmrestart(1,noahrst)
call registerlsmoutput(1,noah_output)
call registerlsmf2t(1,noah_f2t)
call registerlsmwrst(1,noah_writerst)
```

The registry functions defined for noah are:

`noah_varder_ini`	Initialization for Noah
`noah_setup`	Sets up Noah's parameters
`noah_dynsetup`	Sets up Noah's time dependant parameters
`noah_main`	Runs the Noah model on a single gridcell at a timestep
`noahrst`	Reads the Noah restart files
`noah_output`	Writes output of Noah runs
`noah_writerestart`	Writes Noah's restart files
`noah_f2t`	Transfers forcing data to Noah model tiles

The index used for Noah's functions in this case is 1. The index needs to be the same for all registry defintions for a particular model. The user, however, may define any integer value as the index chosen for a land surface model. The corresponding index needs to be specified in the lis card file (`LIS%d%LSM`); i.e., for this example, if the Noah model is used, `LIS%d%LSM` should be assigned a value of 1. [1]

The following code segment shows an example of defining two different land models (CLM and Noah) to be included in the LIS executable. The same procedure can be extended to define more models, or customize LIS to use only the models of interest.

```
call registerlsmini(1,noah_varder_ini)
call registerlsmini(2,clm_varder_ini)

call registerlsmsetup(1,noah_setup)
call registerlsmsetup(2,clm2_setup)

call registerlsmdynsetup(1,noah_dynsetup)
call registerlsmdynsetup(2,clm2_dynsetup)

call registerlsmrun(1,noah_main)
call registerlsmrun(2,driver)

call registerlsmrestart(1,noahrst)
call registerlsmrestart(2,clm2_restart)

call registerlsmoutput(1,noah_output)
call registerlsmoutput(2,clm2_output)

call registerlsmf2t(1,noah_f2t)
call registerlsmf2t(2,atmdrv)

call registerlsmwrst(1,noah_writerst)
call registerlsmwrst(2,clm2wrst)
```

[1] The LIS development team has assigned Noah = 1, CLM = 2, and VIC = 3 for the `LIS%d%LSM` variable. While these numbers may be over-ridden, care must be taken. LIS' documentation refers to this variable as thus stated.

The run-time specific parameters for a land surface model can be read in at run-time through the LIS card file. The user needs to specify a customized namelist and provide routines for reading the same. For Noah runs, the LIS card file contains a namelist segment such as:

```
&noah
noahdrv%WRITEINTN    = 3
noahdrv%NOAH_RFILE   = "noah.rst"
noahdrv%NOAH_MGFILE  = "/X6RAID/MODIS-0.25/"
noahdrv%NOAH_ALBFILE = "/X6RAID/MODIS-0.25/"
noahdrv%NOAH_VFILE   = "BCS/noah_parms/noah.vegparms.txt"
noahdrv%NOAH_SFILE   = "BCS/noah_parms/noah.soilparms.txt"
noahdrv%NOAH_MXSNAL  = "/X6RAID/MODIS-0.25/maxsnalb.bfsa"
noahdrv%NOAH_TBOT    = "/X6RAID/MODIS-0.25/tbot.bfsa"
noahdrv%NOAH_ISM     = 0.30
noahdrv%NOAH_IT      = 290.0
noahdrv%NOAH_NVEGP   = 7
noahdrv%NOAH_NSOILP  = 10
/
```

The namelist specifies variables such as locations of land surface model specific parameter files, output writing intervals, initial conditions, etc. The routine to read these variables is typically done during initialization of the land surface model. The program segment for Noah is shown below as an example. For an explanation of other routines, please refer to the source code documentation.

```
!---------------------------------------------------------------
! Reads the Noah name list
!---------------------------------------------------------------
  if ( masterproc ) then
       call readnoahcrd(noahdrv)
  endif
!---------------------------------------------------------------
! Defines the derived types for MPI, and broadcasts the
! namelist variables to all processors. This step can be
! skipped if using a sequential execution.
!---------------------------------------------------------------
    call def_noahpar_struct
    call MPI_BCAST(noahdrv, 1, MPI_NOAHDRV_STRUCT, 0, &
         MPI_COMM_WORLD, ierr)
!--------------------------------------------------------------
! Allocates Memory for Noah variables
!--------------------------------------------------------------
    if ( masterproc ) then
       allocate(noah(nch))
```

```
else
   allocate(noah(di_array(iam)))
endif
```

The *src/driver* directory contains a number of modules that provides helpful variables that may be required while defining land surface model specific routines. Some of the useful modules and the variables provided by them are listed below. For more details, please refer to the source code documentation.

Module name	Provides
time_manager	Variables and routines for time management
lisdrv_module	`grid`: Vector representation of the running domain grid `tile`: Vector representation of the running domain tiles `gindex`: Mapping of the running domain grid to the corresponding 2-D grid
grid_spmdMod	Variables and routines that define domain decomposition of the vector grid space
tile_spmdMod	Variables and routines that define domain decomposition of the vector tile space
def_ipMod	Variables and routines that are required to carry out spatial interpolation of scalar data

6 Customizing LIS to use new forcing schemes

The boundary conditions describing the (upper) atmospheric fluxes are known as "forcings". LIS makes use of model derived data as well as satellite and ground-based observational data as forcings. The land surface models are typically run using model derived data. The observational data is used to overwrite the model derived data, whenever they are available. LIS driver provides interfaces to incorporate model derived (base) forcing schemes as well as observational (currently for radiation and precipitation products) forcing schemes.

The *forcing-plugin* directory contains modules *baseforcing_pluginMod*, *precipforcing_pluginMod*, and *radfocing_pluginMod*, that can be used to customize and define base forcing schemes, observed precipitation forcing schemes, and observed radiation forcing schemes, respectively. These modules provide plugin routines *baseforcing_plugin*, *precipforcing_plugin*, and *radforcing_plugin*, respectively.

baseforcing_module provides registries to define functions to perform the following tasks.

- definition of native domain:
  Routines to define the native domain of the forcing data, read run-time specific parameters through a namelist, etc.
- retrieval of forcing data:
  Routines to retrieve the forcing data, and interpolate them.
- temporal interpolation:
  Routines to interpolate data temporally.

The following code segment shows how two baseforcing schemes are included in LIS.

```
call registerdefnat(1,defnatgdas)
call registerdefnat(2,defnatgeos)

call registerget(1,getgdas)
call registerget(2,getgeos)

call registertimeinterp(1,time_interp_gdas)
call registertimeinterp(2,time_interp_geos)
```

Similar to the case in *lsm_pluginMod*, the indices used in the registries need to be consistent for a particular scheme. In the example shown above, the GDAS forcing scheme is assigned index 1, and the GEOS forcing scheme is assigned 2. These indices are arbitrary, but the indices used in the card file (`LIS%d%FORCE`) should reflect the ones defined in the registry. i.e., if GEOS forcing scheme is to be used, `LIS%d%FORCE` should be assigned a value of 2. [2]

[2]The LIS development team has assigned GDAS = 1, and GEOS = 2 for the `LIS%d%FORCE` variable. While these numbers may be over-ridden, care must be taken. LIS' documentation refers to this variable as thus stated.

The run-time specific parameters for a forcing scheme can be specified at run-time through the lis card file. The user needs to specify a customized namelist and provide routines for reading the same. For GEOS runs, the section of the lis card file contains a namelist segment such as:

```
&geos
geosdrv%GEOSDIR    = "/X6RAID/DATA/GEOS/BEST_LK"
geosdrv%NROLD      = 181
geosdrv%NCOLD      = 360
geosdrv%NMIF       = 13
/
```

The namelist specifies variables such as locations of forcings files, the native domain sizes, the number of variables in each file, etc. The routine to read these parameter is done typically while defining the native domain parameters of the forcing scheme. A sample routine for GEOS forcing scheme is shown below.

```
!---------------------------------------------------------------
! Reads the GEOS name list
!---------------------------------------------------------------
    call readgeoscrd(geosdrv)
!---------------------------------------------------------------
! Defines the native GEOS domain as a kgds array
!---------------------------------------------------------------
    kgdsi(1) = 0
    kgdsi(2) = geosdrv%ncold
    kgdsi(3) = geosdrv%nrold
    kgdsi(4) = -90000
    kgdsi(7) = 90000
    kgdsi(5) = -180000
    kgdsi(6) = 128
    kgdsi(8) = 179000
    kgdsi(9) = 1000
    kgdsi(10) = 1000
    kgdsi(20) = 255
    mi = geosdrv%ncold*geosdrv%nrold
```

precipforcing_module provides registries to define functions to perform the following tasks.

- definition of native domain:
  Routines to define the native domain of the forcing data, read run-time specific parameters through a namelist, etc.

- retrieval of forcing data:
  Routines to retrieve forcing data and interpolate them.

The following code segment shows how the CMAP precipitation scheme is included in LIS.

```
call registerdefnatpcp(4,defnatcmap)
call registerpget(4,getcmap)
```

The indexing scheme is similar to the cases described above. In this case, the CMAP scheme is assigned an index of 4. `LIS%f%GPCPSRC` in the card file should correspond to the indices defined in the registries. A value of 0 indicates that no observed precipitation scheme will be employed.

The customized namelist section for CMAP is shown below.

```
&cmap
cmapdrv%CMAPDIR    = "./input/CMAP"
cmapdrv%NROLD      = 181
cmapdrv%NCOLD      = 360
/
```

The namelist specifies variables such as locations of forcings files, the native domain sizes, etc. The routine to read these parameter is done typically while defining the native domain parameters, similar to the base forcing case.

The design of *radforcing_module* is similar to the cases described above. The registry functions for this module are:

- definition of native domain:
  Routines to define the native domain.
- retrieval of forcing data:
  Routines to retrieve and interpolate data.
- Interpolate data in time:
  Temporallly interpolate data.

An example of using AGRMET observed radiation scheme is shown below.

```
call registerrget(1,getgrad)
call registerdefnatrad(1,defnatagrmet)
call registerrti(1,time_interp_agrmet)
```

The indices defined for observed radiation schemes correspond to the `LIS%f%RADSRC` in the card file. The value is defined to be 0 if no observed radiation schemes are used.

As mentioned earlier, the modules in *src/driver* can be used in defining routines needed for defining a forcing scheme in LIS. Please refer to the source code documentation for details.

7 Customizing LIS for a new domain

The LIS driver is designed to be domain independent. The parameters used to define the domain are designed to be run-time options. The user is required to specify two different types of domain information

- Running domain:
  The domain over which land surface simulations are carried out.
- Data domain:
  The domain over which the parameter data-sets are defined. Currently it is assumed that all data-sets are defined in the same domain. In future, support for defining domain information for each dataset will be provided.

The parameters used to define a domain are adopted from the design used in grib decoding programs such as w3fi63 [8]. The *domain* namelist in the lis card file specifies an array called `kgds`. Indices from 1 to 20 in the `kgds` array define the running domain, and indices from 41 onwards define the parameter data domain. The description of `kgds` array is shown below for the running domain. The parameter data domain can be defined in a similar way. The following sections describe the `kgds` array used in the card file for the running domain.

Variable	Description
`LIS%d%kgds(1)`	0 - Latitude/Longitude
	1 - Mercator cylindrical
	3 - Lambert conformal conical
	4 - Gaussian cylindrical
	5 - Polar stereographic azimuthal
For latitude, longitude grids,	
`LIS%d%kgds(2)`	Number of points on a latitude circle
`LIS%d%kgds(3)`	Number of points on a longitude circle
`LIS%d%kgds(4)`	Latitude of origin
`LIS%d%kgds(5)`	Longitude of origin
`LIS%d%kgds(6)`	Resolution flag
`LIS%d%kgds(7)`	Latitude of extreme point
`LIS%d%kgds(8)`	Longitude of extreme point
`LIS%d%kgds(9)`	Latitudinal directional increment
`LIS%d%kgds(10)`	Longitudinal directional increment
`LIS%d%kgds(11)`	Scanning mode flag

The LIS driver currently supports Latitude/Longitude.The support for other types of grids are in development. The user is expected to provide parameter data that is consistent with the domain specified at run-time. The details of specifying parameter data is explained in the next section.

7.1 KGDS Example

This section describes how to compute the values for the kgds array.

First, we shall generate the values for the parameter data domain. These are the values for kgds(41) – kgds(50). LIS' parameter data is defined on a Latitude/Longitude grid, from -60 to 90 degrees latitude and from -180 to 180 degrees longitude.

Since the parameter data is on a Latitude/Longitude grid, we have

```
LIS%d%kgds(41)    = 0
LIS%d%kgds(46)    = 128
```

For this example, consider running at 1/4 deg resolution. The coordinates of the origin and extreme point are specified at the grid-cell center. Here the "origin" grid-cell is given by the box (-60,-180), (-59.750,-179.750). The center of this box is (-59.875,-179.875). Since the values of the origin point must be integers scaled by 1000, we have

```
LIS%d%kgds(44)    = -59875
LIS%d%kgds(45)    = -179875
```

The "extreme" grid-cell is given by the box (89.750,179.750), (90,180). Its center is (89.875,179.875). Scaling these by 1000 gives,

```
LIS%d%kgds(47)    = 89875
LIS%d%kgds(48)    = 179875
```

Scaling the resolution (0.25 deg) by 1000 gives

```
LIS%d%kgds(49)    = 250
LIS%d%kgds(50)    = 250
```

The number of points on a longitude circle or a latitude circle is given by

$$\frac{\texttt{Extreme point} - \texttt{Origin point}}{\texttt{resolution}} + 1$$

Thus the number of points on a longitude circle are

$$\frac{179875 - -179875}{250} + 1 = 1440$$

Giving,

```
LIS%d%kgds(42)    = 1440
```

And the number of points on a latitude circle are

$$\frac{89875 - -59875}{250} + 1 = 600$$

Giving,

```
LIS%d%kgds(43)    = 600
```

And this completely defines the parameter data domain.

If you wish to run over the whole domain defined by the parameter data domain then you simply set the values of kgds(1) – kgds(10) equal to the values given by kgds(41) – kgds(50). This gives

```
LIS%d%kgds(1)     = 0
LIS%d%kgds(2)     = 1440
LIS%d%kgds(3)     = 600
LIS%d%kgds(4)     = -59875
LIS%d%kgds(5)     = -179875
LIS%d%kgds(6)     = 128
LIS%d%kgds(7)     = 89875
LIS%d%kgds(8)     = 179875
LIS%d%kgds(9)     = 250
LIS%d%kgds(10)    = 250
```

Now say you wish to run only over the region given by (27.8,-97.6), (31.9,-92.8). Since the running domain is a sub-set of the parameter domain, it is also a Latitude/Longitude domain at 1/4 deg. resolution. Thus,

```
LIS%d%kgds(1)     = 0
LIS%d%kgds(6)     = 128
LIS%d%kgds(11)    = 64
LIS%d%kgds(9)     = 250
LIS%d%kgds(10)    = 250
```

Now, since the running domain must fit onto the parameter domain, the desired running region must be expanded from (27.8,-97.6), (31.9,-92.8) to (27.75,-97.75), (32.0,-92.75). The "origin" grid-cell for the running domain is the box (27.75,-97.75), (28.0,-97.5). Its center is (27.875,-97.625). Scaling this by 1000 gives

```
LIS%d%kgds(4)     = 27875
LIS%d%kgds(5)     = -97625
```

The "extreme" grid-cell for the running domain is the box (31.75,-93), (32.0,-92.75). Its center is (31.875,-92.875). Scaling this by 1000 gives

```
LIS%d%kgds(7)     = 31875
LIS%d%kgds(8)     = -92875
```

Using the above equation for the number of points in a longitude circle, we get

$$\frac{-92875 - -97625}{250} + 1 = 20$$

giving

```
LIS%d%kgds(2)     = 20
```

Similarly, the number of points in a latitude circle is

$$\frac{31875 - 27875}{250} + 1 = 17$$

giving

```
LIS%d%kgds(3)     = 17
```

This completely defines the running domain.

8 Customizing LIS for Input data

LIS allows the user to specify a number of input data-sets at run-time through the card file. These data-sets are expected to be consistent with the type of domain and resolution specified. Land model specific parameter data are expected to be specified in the namelists specific to each land model. The input data for models in LIS are divided into three categories:

- Soils data: static
- Vegetation data: some static and some dynamic
- Meterological data: at different frequencies

The soil and vegetation data are used to specify the characteristics of the land surface and the meterological data to provide forcing at the upper boundary of the land surface. A detailed description of parameter data used in LIS are available at `http://lis.gsfc.nasa.gov/Data/index.shtml`.

8.1 Soils data

LIS provides a number of overlapping data-sets for specifying soil hydraulic properties: sand, clay, silt, and organic texture fractions, porosity maps, etc.

The following labels describe the format the data files are written in.

bfsa	- binary, sequential access
txt	- text
bfda	- binary, direct acess

Data file	Description	Format
`LIS%p%SAFILE`	Sand fraction map file	bfsa
`LIS%p%CLFILE`	Clay fraction map file	bfsa
`LIS%p%ISCFILE`	Soil color map file	bfsa
`LIS%p%ELEVFILE`	Elevation difference file	bfsa

Some of the model specific parameter data are specified below:

`noahdrv%noah_sfile`	Noah soil parameter file	txt
`vicdrv%vic_sfile`	VIC soil parameter file	txt

8.2 Vegetation data

LIS uses a number of files to specify both static and time-varying vegetation properties. Some of the files are:

`LIS%p%MFILE`	Land/Water map file for modeling	bfsa
`LIS%p%VFILE`	Vegetation classification map file	bfsa
`LIS%p%AVHRRDIR`	Location of AVHRR-based LAI/SAI files	bfda
`LIS%p%MODISDIR`	Location of MODIS-based LAI/SAI files	bfda

CLM

`clmdrv%clm2_vfile`	CLM mapping from UMD to plant functional types	txt

Noah

`noahdrv%noah_mgfile`	Location of monthly veg. greenness fraction	bfsa
`noahdrv%noah_albfile`	Location of quarterly snow free albedo	bfsa
`noahdrv%noah_vfile`	Noah static vegetation parameter file	txt
`noahdrv%noah_mxsnal`	Maximum snow free albedo	bfsa
`noahdrv%noah_tbot`	Bottom temperature	bfsa

VIC

`vicdrv%vic_veglibfile`	VIC vegetation parameter file	txt

8.3 Meterorological data

LIS includes a number of forcing schemes, both model-derived as well as observation based. A summary of the forcing data schemes implemented in LIS are shown below.

Forcing scheme	Type	Frequency
GEOS	model derived	3 hourly
GDAS	model derived	3 hourly
AGRMET	observational (radiation)	hourly
CMAP	observational (precipitation)	3 hourly

Implementation of other forcing schemes is currently under development.

References

[1] Community climate system model, software developers guide. http://www.ccsm.ucar.edu/csm/working_groups/Software/dev_guide/dev_guide/.

[2] Community land model. http://www.cgd.ucar.edu/tss/clm.

[3] GrADS. http://grads.iges.org/grads/grads.html.

[4] Noah land surface model. http://www.emc.ncep.noaa.gov/mmb/gcp/noahlsm/README_2.2.htm.

[5] Protex documentation system. http://gmao.gsfc.nasa.gov/software/protex.

[6] DODS. http://www.unidata.ucar.edu/packages/dods/.

[7] Variable infiltration capacity (vic) model. http://www.hydro.washington.edu/Lettenmaier/Models/VIC/VIChome.html.

[8] W3fi63 program. http://dss.ucar.edu/datasets/ds609.1/software/mords/w3fi63.f.

[9] G. J. Collatz, C Grivet, J. T. Ball, and J. A. Berry. Physiological and environmental regulation of stomatal conducatance: Photosynthesis and transpiration: A model that includes a laminar boundary layer. *Agric. For. Meteorol.*, 5:107–136, 1991.

[10] Chen. F., Mitchell. K., Schaake. J, Xue. J, Pan. H, Koren. V., Ek. M Duan, and A. Betts. Modeling of land-surface evaporation by four schemes and comparison with fife observations. *J. Geophys. Res.*, 101(D3):7251–7268, 1996.

[11] P. G. Jarvis. The interpretation of leaf water potential and stomatal conductance found in canopies of the field. *Phil. Trans. R. Soc.*, 273:593–610, 1976.

[12] L. A. Richards. Capillary conduction of liquids in porous media. *Physics*, 1:318–333, 1931.

[13] E. Rogers, T. L. Black, D. G. Deaven, G. J. DiMego, Q. Zhao, M. Baldwin, N. W. Junker, and Y. Lin. Changes to the operational "early" eta analysis/forecast system at the national centers of environmental prediction. *Wea. Forecasting*, 11:391–413, 1996.